Fermilab/BD/TEV
Beams-doc-860-v0
October 2, 2003
Version 0.0

Tevatron Beam Position Monitor Software Specifications

DRAFT DRAFT DRAFT

Margaret Votava, Luciano Piccoli, Dehong Zhang
Fermilab, Computing Division, CEPA

Brian Hendricks
Fermilab, Beams Division, Accelerator Controls Department

Abstract

Specification of BPM front-end software.

Table of Contents

Overview

This note documents accumulated knowledge about the software needed for the data acquisition software needed for the TEV BPM upgrade project. The data acquisition software will run on VME the front-end computers. Figure 1 shows the software structure and the different elements involved with the BPM upgrade project.

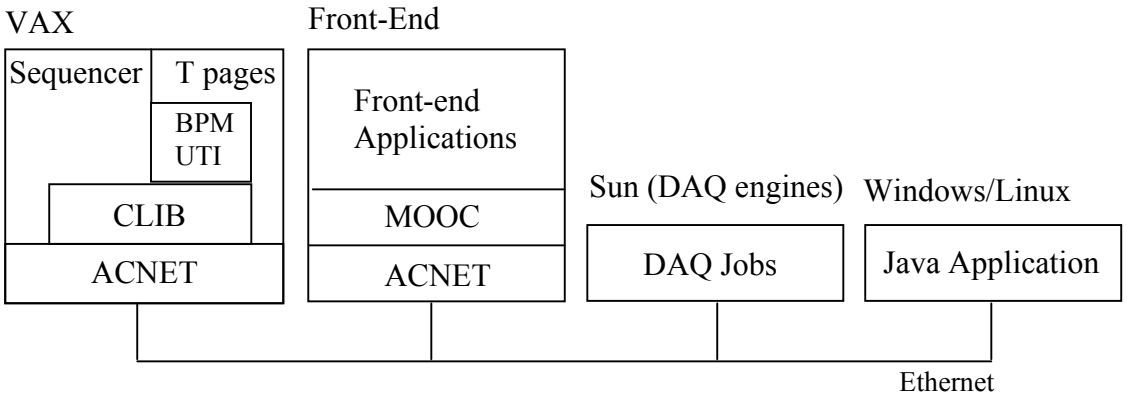

Figure 1 – Overall software architecture

The DA software is the portion of code that resides on the front-end microprocessors. For compatibility reasons and to limit the amount of effort needed on the online front, the new DA software will need to have several constraints imposed by the existing architecture. These include:

From the online application perspective, all communication with the front ends is via ACNET devices. This includes data readout as well as setting readout parameters. Internal diagnostics, however, do not have this constraint.

Data collection happens asynchronously from data readout, ie, the BPMs can be configured to take data continuously on certain triggers, but the data is not read out until later. Not all data that is collected is readout. This implies that the DA must manage readout requests from the online software.

"Event assembly" is done by the online software, not the DA, therefore a given BPM house does not need to have any knowledge about any other BPM house(s).

Any real-time OS that is needed will be VxWorks

There is one VME processor running the front-end DA, which will be responsible for handling multiple BPM cards.

Data Acquisition

The BPM front-end reads data from the BPM hardware and passes it on to the online applications. The front-end readout events are armed by TCLK events or state device transitions and triggered by BSYNCH signals. There is the option of having programmable delays for triggering.

Data is always being read out from the BPM hardware; depending on the hardware configuration the front-end will have to switch between data acquisition modes in order to process users requests. If mode-switching is not required by hardware, all information about orbits and single turns and turn by turn can be made available simultaneously.

The front-end will keep internal buffers that can be read using ACNET devices, more specifically the **protocols** setdat and retdat, fast time plot and snapshot (not to be confused with BPM snapshot).

Data Acquisition Modes

The BPMs are capable or collecting the data in a few different modes. What of these is done in the hardware/firmware, and what is done by the controller? EG, the controller can have enough memory to keep several buffered closed orbits if the time between triggers is long enough not to overload the CPU. Is the DA mode mutually exclusive of other modes?

Single Closed Orbit - Based on average beam position. The average is usually based on 8, 16, 32 or 64 single turn measurements. The front-end software should also be able to receive and process requests for averaging closed orbits over any number of turns (for the purpose of averaging out synchrotron oscillations).

Buffered Closed Orbit – An array of closed orbit measurements. This array can be built by a repeating trigger of a specific type. Time interval too? Can be built by multiple triggers? How many of these can happen in parallel? Need a repeating one for quench post mortems.

Single Turn - Position of a single pass of the beam (first bunch) through the BPM. The BPMs must get the position of the same bunch for the same revolution.

Turn By Turn - Collection of *N* consecutive single turn measurements.

Output Data

The data sent from the front-end DAQ to the BPM library and/or applications is based in the following C data structures. For compatibility, the BPM libraries will be responsible

for extracting subsets from these data, which are required by existing application programs.

Generic Headers

```
typedef struct BPM_TIME {
    ulong timestamp;          /* timestamp in seconds (GMT) */
    ulong nanoseconds;        /* nanoseconds */
}

typedef               struct               TRIGGER_INFO               {
    to be defined;
}

typedef struct TEVATRON_BPM_HEADER {
    long endian_type;         /* 0    -> little endian,
                                 else -> big endian */
    long version;             /* data structure version */
    long status;              /* overall status: zero = OK */
    BPM_TIME time;            /* time stamp */
    ulong turn_number;        /* starting turn number */
    ulong num_turns;          /* number of turns in data */
    double time_in_cycle;     /* starting time in cycle */
    long data_type;           /* flash/snapshot/profile/TBT/etc */
    TRIGGER_INFO trigger_info;/* trigger information */
    long data_source;         /* 0 -> beam,
                                 1 -> calibration system,
                                 2 -> software diag,
                                 3 -> hardware diag */
    long particle_type;       /* 0 -> proton, 1 -> pbar */
    long bunch_type;          /* 0 -> uncoalesced, 1 -> coalesced */
    long scaled_data;         /* 0 -> raw data, 1 -> scaled data */
    long machine_state;       /* value of V:CLDRST */
    long helix_state;         /* 0 -> helix off, 1 -> helix on */
}
```

BPM Non Turn By Turn

```
typedef struct TEVATRON_BPM_FRAME_DATA {
    long frame_number;        /* ordinal number in front end */
    BPM_TIME time;            /* time stamp */
    ulong turn_number;        /* starting turn number */
    double time_in_cycle;     /* starting time in cycle */
    long num_detectors;       /* number of detectors present */
    float positions[12];      /* position values in mm */
    float intensities[12];    /* intensity values */
    char status[12];          /* status values */
}
```

Proposed status values:

```
OK = 0
invalid reading = 1 (too little beam intensity?)
alarm level = 3 (if we want alarm limits)
saturated = 5
error = -1 (error reading value (hardware error?))
unequipped = -2 (channel is not in use)
```

This would be the final structure of the ACNET device for display, snapshot, profile, and flash frames.

```
typedef struct TEVATRON_BPM_ORBIT_DATA {
    TEVATRON_BPM_HEADER header;
    long num_frames;    /* number of frames returned */
    TEVATRON_BPM_FRAME_DATA frame_data[];
}
```

BPM Turn By Turn

```
typedef struct TEVATRON_BPM_TBT_TURN {
    unsigned long turn_number;/* turn number */
    float position;           /* position in mm */
    float intensity;          /* beam intensity */
}
```

This would be the final structure of the ACNET device for turn by turn data.

```
typedef struct TEVATRON_BPM_TBT_DATA {
    TEVATRON_BPM_HEADER header;
    short status;       /* detector status */
    long num_turns;     /* number of turns returned */
    TEVATRON_BPM_TBT_TURN turn_data[];
}
```

BLM Data Structures

```
typedef struct TEVATRON_BLM_FRAME_DATA {
    long frame_number;        /* ordinal number in front end */
    BPM_TIME time;            /* time stamp */
    ulong turn_number;        /* starting turn number */
    double time_in_cycle;     /* starting time in cycle */
    long num_detectors;       /* number of detectors present */
    float losses[24];         /* loss values in rads/sec */
    char status[24];          /* status values */
}
```

Proposed status values:

```
OK = 0
alarm level = 3 (if we want alarm limits)
saturated = 5
error = -1 (error reading value (hardware error?))
unequipped = -2 (channel is not in use)
```

This would be the final structure of the ACNET device for display, snapshot, profile, and flash frames.

```
typedef struct TEVATRON_BLM_DATA {
    TEVATRON_BPM_HEADER header;
    long num_frames;    /* number of frames returned */
    TEVATRON_BLM_FRAME_DATA frame_data[];
}
```

Data Consumers

Some of the data may have multiple [online] consumers. Since the request for an event happens asynchronously from reading out of the data from the BPM, how do we ensure

that the correct data is being sent to the correct consumer? Some consumers may be requesting the same event.

The software must be able to support fast time plots (FTP). Data will be requested by consumers at a rate of up to 500 Hz.

Goals

It is a goal to have the BPMs detect state changes via the state devices (in the old system, the sequencer would notify the BPMs of changes). This will help to reduce the complexity of the sequencer, allow for faster builds and testing of BPM changes, and push the knowledge of BPM behavior to the BPMs themselves.

State Diagram

Please refer to the Tevatron States Device and Clock Events URL: http://www-bdnew.fnal.gov/tevatron/adcon/tev_states.html for a detailed description of the tevatron state diagram.

State Devices of interest to the BPMs are:

V:CLDRST – state of the Tev Collider
V:TEVMOD – tevatron high-level mode (colliding beams, protons only, dry squeeze, ramping, recovery/turn on, off)
V:COALP – proton beam type (coalesced or uncoalesced)
V:COALA – pbar beam type (coalesced or uncoalesced)
V:TVBEAM – beam status (no beam, protons, pbars or protons and pbars)

BPMs need to provide the following functionality through the various states (V:CLDRST)

Proton Injection Porch

From the time the tevatron is in *recovery (no TCLK event?)* until it reaches either the *proton injection porch* (event $43), the BPMs are not required to be doing anything. Does an inventory of the BPMs – are they working etc. What will be the communication path to notify the smoothing programs of bad BPMs? What type of data is taken here?

Proton Injection Tuneup

When the TCLK event $2B + $4D is received, the BPMs should mark themselves in batch mode (uncoalesced). Only proton data is needed in this state. Take both single turn, in particular, first turn, as well as triggered (either on a TCLK event or a TCLK event + delay) closed orbit data in this state. No pbar data available.

Reverse Injection

Currently only uses Main Injector BPM data taking closed orbit data triggered on a TCLK.

Inject Protons

Switch now to coalesced beam depending on the operational mode (we can tell this from V:TEVMOD ? Check that from V:COALP). SDA is collecting closed orbit data on demand. May want this to be triggered?

Pbar Injection Porch

SDA collecting proton closed orbit data on demand.

Inject Pbar

Both p and pbar data needed here. Collecting closed orbit data on TCLK events as well as on demand. There may be unique cogging values for each BPM. What BPM data is needed to generate this? How and when is the cogging information loaded?

Acceleration

Profiling data taken. Profiling is a closed orbit measurement taken at *n* points during the acceleration process at 100GeV, 200GeV,... , 980GeV. Right now p only – pbar would be nice. Currently generated by a TCLK event that is coming from the CAMAC timing modules. Could be done with internal timers?

ACNET devices

The BPMs will use the following ACNET devices:

XYZ[0]

Do they need their own state devices?

Are there pbar devices for all corresponding p devices?

Calibration

Calibration constants need to be received and sent down to the DSPs. Calibration data can either be sent automatically upon state device changes or manually. What is the calibration data? How often it is downloaded and when?

Diagnostics

The DAQ software will provide diagnostics data both via ACNET devices and using other protocol over ethernet. It will provide means for operators or programs to detect a bad or misbehaving BPM.

The preferred protocol for implementing diagnostics is sockets over TCP/IP. Each front-end will provide standard ports for requesting tests, and the results can either be made available through ACNET devices, such that test values can be read by the VAX applications, or sent via sockets to diagnostics programs running on the Sun machines.

Some diagnostics operations follow:

- Generate close orbit: return known closed orbit values via ACNET and/or sockets.

- Generate turn by turn: return known values for a turn by turn measurement via ACNET and/or sockets.
- Generate single turn: return known values for a single turn measurement via ACNET and/or sockets.
- Check BPM hardware: run test procedures in the BPM hardware (one or all the BPM boards) – if supported by the hardware. Return findings via ACNET and/or sockets.
- Check BLM hardware: run test procedures in the BLM hardware (one or all the BLM boards) – if supported by the hardware. Return findings via ACNET and/or sockets.
- Get buffers: return current contents of all (or selected) data buffers via socket.
- Set ACNET devices to known values: specify values to be set in the ACNET devices.

Self-Testing Procedures

The front-end DAQ should be able to perform tests on itself and on the associated BPM hardware. Results from self-tests should be available to user applications.

Hardware tests will be performed if supported, i.e., the hardware should have the capability of receiving triggers from the front-end and generate data for self-tests.

Monitoring

The front-end DAQ should periodically send status and statistics messages to a monitor, via either ACNET devices or socket connections. There should be a central monitoring application which receives data from all BPM front-ends.

Data from the front-end include:

- Buffer usage
- Up time
- Available memory
- Status of processes
- Number of requests
- Tevatron status

Configuration Parameters

Name	Description	When can Change	Range
Sample Size	Number of samples used in closed orbit averaging	Proton Injection Tuneup	N - M
Operational Mode Can these be	Define the current mode of operation of the BPMs	On state device changes and on manual request	Closed Orbit, Single Turn or Turn by Turn

done in parallel?			

Glossary

Time Line Generator (TLG) – one [of several] of the TCLK sources. The TLG is a continuous cycle sends out various TCLK signals in a periodic fashion. The particular TCLK signals it sends is a function of its mode which in turn is set by the sequencer. An example mode is xzy which would generate the TCLK signals for xyz.